SCIENCE CHINA
Information Sciences

• REVIEW •

February 2015, Vol. 58 021101:1–021101:24
doi: 10.1007/s11432-014-5241-2

A survey on bug-report analysis

ZHANG Jie[1], WANG XiaoYin[2], HAO Dan[1*], XIE Bing[1], ZHANG Lu[1*] & MEI Hong[1]

[1]*Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education, Beijing* 100871*, China;*
[2]*Department of Computer Science, University of Texas at San Antonio, San Antonio, USA*

Received April 10, 2014; accepted September 12, 2014

Abstract Bug reports are essential software artifacts that describe software bugs, especially in open-source software. Lately, due to the availability of a large number of bug reports, a considerable amount of research has been carried out on bug-report analysis, such as automatically checking duplication of bug reports and localizing bugs based on bug reports. To review the work on bug-report analysis, this paper presents an exhaustive survey on the existing work on bug-report analysis. In particular, this paper first presents some background for bug reports and gives a small empirical study on the bug reports on Bugzilla to motivate the necessity for work on bug-report analysis. Then this paper summaries the existing work on bug-report analysis and points out some possible problems in working with bug-report analysis.

Keywords survey, bug report, bug-report analysis, bug-report triage, bug localization, bug fixing

Citation Zhang J, Wang X Y, Hao D, et al. A survey on bug-report analysis. Sci China Inf Sci, 2015, 58: 021101(24), doi: 10.1007/s11432-014-5241-2

1 Introduction

As programmers can hardly write programs without any bugs, it is inevitable to find bugs and fix bugs in software development. Moreover, it is costly and time-consuming to find and fix bugs in software development. Software testing and debugging is estimated to consume more than one third of the total cost of software development [1,2].

To guarantee the quality of software efficiently, many projects use bug reports to collect and record the bugs reported by developers, testers, and end-users. Actually, with the rapid development of software (i.e., especially open source software), vast amounts of bug reports have been produced. For example, there are on average 29 reports submitted each day in the open-source version of Eclipse[1)] [3]. Although a large number of bug reports may help improve the quality of software, it is also a challenge to analyze these bug reports. To address this practical problem, many researchers proposed various techniques to facilitate bug-report[2)] analysis.

* Corresponding author (email: haodan@pku.edu.cn, zhanglucs@pku.edu.cn)

1) https://bugs.eclipse.org/bugs/.

2) In this paper, we focus on bug reports rather than issue reports. Usually a bug report refers to a software document that describes software bugs, whereas an issue report refers to a software document that describes various issues (e.g., bugs, missing documentation and requested features) of a software. See Section 2 for more details.

Most work on bug-report analysis focuses on how to use bug reports efficiently. As a bug report records various information of a bug, including its stakeholders, description, and so on, some work focuses on how to optimize bug reports [4–6] by addressing the problems in existing bug reports, e.g., misclassifying a new feature as a bug and assign severity levels to bug reports. To collect and assign bug reports automatically, some work focuses on how to automate bug-report triage [3,7–11] by identifying duplicate bug reports on the same bug, predicting the priority of bug reports, and assigning bug reports with high precision. Furthermore, to help developers fix the bugs reported by bug reports, some work focuses on fixing bugs based on these bug reports.

To review these existing work, in this paper, we present the first survey of the current research on bug-report analysis. According to the life-cycle of a bug report, a bug report has at least two stakeholders: the ones who submit bug reports (i.e., reporters) and the ones who deal with bug reports (i.e., developers). We classify the existing work on bug-report analysis based on the stakeholders of bug reports. Some work on bug-report analysis focuses on aiding the ones who submit bug reports by improving the quality of bug reports, i.e., **bug report optimization**, which includes the studies on the quality of bug reports, identifying misclassified bug reports, and predicting the severity of bug reports. Some work on bug-report analysis focuses on aiding the ones who deal with bug reports, i.e., usage of bug reports. In particular, based on the aims of these work, we classify the work on usage of bug reports into two categories, **bug-report triage** and **bug fixing**. Work on bug-report triage aims to triage bug reports automatically, whereas work on bug fixing aims to remove the bugs reported by bug reports. In particular, work on bug-report triage includes bug-report prioritization, duplicate bug-report detection, and bug-report assignment, whereas work on bug fixing includes bug localization based on bug reports and link recovery between bugs and changes. Especially, as a bug report records the document for bugs, many researchers focus on bug fixing so as to reduce developers' manual efforts on removing the bugs reported by bug reports. As most of these bug localization approaches are based on information retrieval techniques, we further classify these approaches based on the data used in information retrieval. For the work on link recovery between bugs and changes, based on the different aims of these work, we classify these work into work on recovery techniques and work on linkage bias revealing.

The rest of this paper is organized as follows. Section 2 presents some terms mostly used in bug-report analysis, the life-cycle of bug reports, and some statistics of existing bug reports in practice, Section 3 further explains our classification framework on bug-report analysis, Section 4 reviews the work on bug-report optimization, Section 5 reviews the work on bug-report triage, Section 6 reviews the work on bug fixing, and Section 7 reviews a little work on other bug-report analysis. Finally, Section 8 concludes the whole paper.

2 Preliminary

In this section, we explain some basic concepts and describe the life-cycle of bug reports. We also provide some statistics as well as analysis of bug reports.

2.1 Bug-report terminology

(1) Bug. A software bug is an error, flaw, or fault in a computer program or system that produces unexpected results or behavior. There is a distinction between "bug" and "issue". An issue could be a bug but not always a bug. It can also be a feature request[3)], task, missing documentation, and so on. The process of finding and removing bugs is called "debugging". Bugs may be caused by tiny coding errors, but the results of bugs can be serious, making finding and fixing bugs a rather challenging task.

(2) Bug report. A bug report is a software document describing software bugs, which is submitted by a developer, a tester, or an end-user [12]. Bug reports have many other names, such as "defect reports" [13], "fault reports", "failure reports" [14], "error reports", "problem reports", "trouble reports", and so

3) The authors of this paper have collected the feature requests of the top 12 projects from sourceforge, which is available at https://github.com/pku-sei-test/Feature-Request.

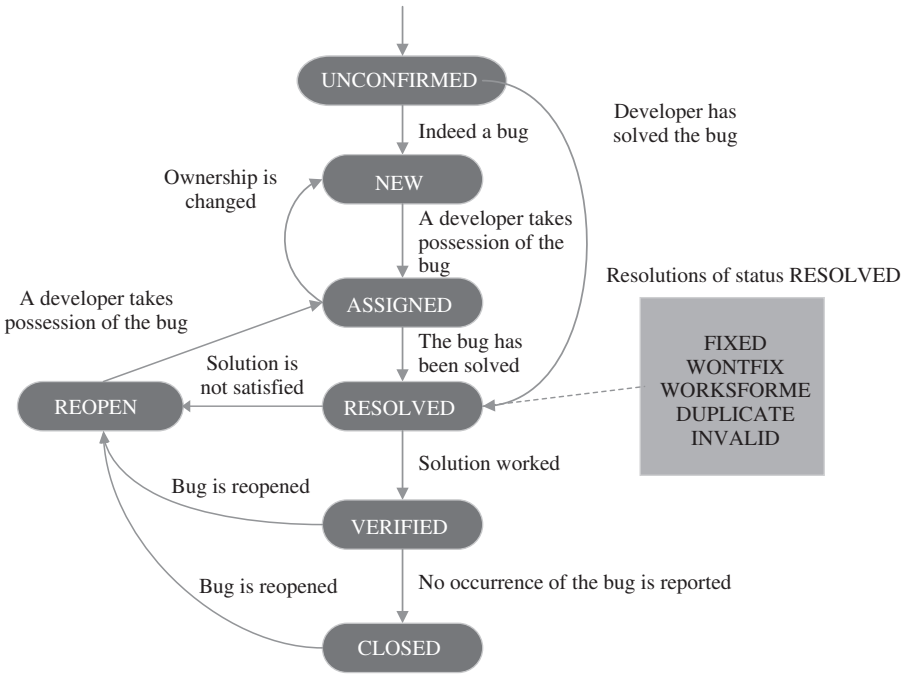

Figure 1 The life-cycle of a bug report

on. Typically a bug report is composed of its identification number, its title, when it is reported and modified, the severity or importance, the programmer assigned to fix the bug, the resolution status (e.g., new, unconfirmed, resolved) of the bug, the description of the report(e.g., steps to reproduce the bug, stack traces, and expected behavior), additional comments (e.g., discussion about the possible solutions), attachments (e.g., proposed patches, test cases), and a list of reports that need to be addressed before this report is resolved [11].

(3) Bug repository. Bug repositories are often used in open-source software projects to allow both developers and users to post problems encountered with the software, suggest possible enhancements and comment upon existing bug reports [10]. An open bug repository is open and visible for all people. As bugs reported in bug repositories may be identified and solved by all people, the quality of the open projects may improve [15].

(4) Bug-report triage. Bug-report triage consists of the following process: figuring out whether a bug is a real bug, checking whether the reported bug is a duplicate of an existing bug, prioritizing bug reports, and deciding which developer should work on the bug reports [16].

(5) Bug-report duplication. Duplicate bug reports refer to the bug reports on the same bug committed by different reporters, as there are many users interacting with a system and reporting its bugs. Detecting duplicate bug reports is a procedure of bug triage, which reduces triaging cost and saves time for developers in fixing the same issues [8].

(6) Bug tracking system. A bug tracking system (also called defect tracking system) manages bug reports and developers who fix bugs [16]. Bug tracking systems are designed to track reported software bugs. Bugs are stored in a bug repository, which is the major component of a bug tracking system. To submit and resolve bugs automatically, developers of many popular open-source projects (e.g., *Mozilla*, *Eclipse*, and *Linux kernel*) use bug tracking systems (e.g., *Bugzilla*, *Jira*, *Mantis*, *Trac*).

2.2 The life-cycle of a bug report

A bug report goes through several resolution status over its lifetime. Figure 1 depicts the life-cycle of a bug report.

When a bug is first reported, the bug report is marked as *UNCONFIRMED*. When a triager has verified

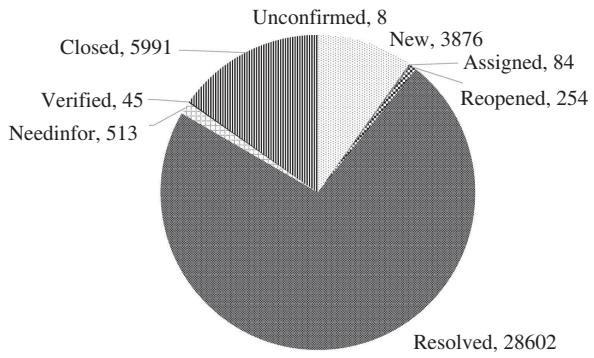

Figure 2 Number of bug reports for Apache.

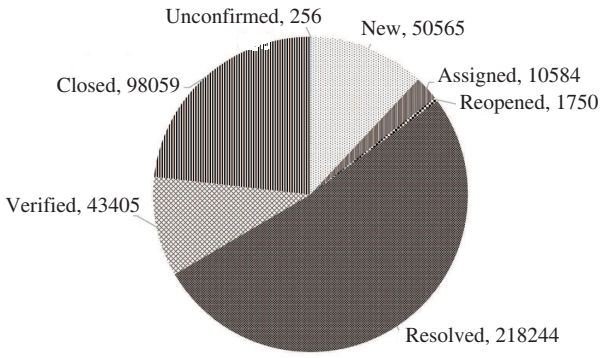

Figure 3 Number of bug reports for Eclipse.

that the bug is not duplicate and indeed a new bug, the status is set to *NEW*. Then the triager assigns the bug report to one proper developer, and the status is changed to *ASSIGNED*. Then the assigned developer reproduces the bug, localizes it and tries to fix it. When the bug has been solved, the bug report is marked as *RESOLVED*. After that, if a tester is not satisfied with the solution, the bug should be reopened with the status set to *REOPEN*; if a tester has verified that the solution worked, the status is changed to *VERIFIED*. The final status of a bug report is *CLOSED*, which is set when no occurrence of the bug is reported.

Note that there are more than one possible sequent status for the status *RESOLVED*. If a developer makes necessary code changes and the bug is solved, the status will be changed from *RESOLVED* to *FIXED*. If the developer does not fix the bug due to some reason, the status will be set to *WONTFIX*. If the problem could not be reproduced by the developer, the status will be changed to *WORKSFORME*. If the developer finds that the report is a duplicate of an existing bug report, the status will be changed to *DUPLICATE*. If the bug reported by the bug report is not an actual bug, the status will be changed to *INVALID*.

2.3 Statistics of bug reports

In this section, we take the most popular bug tracking system—Bugzilla[4)] as a subject and study the bug reports in this system. Bugzilla is a bug tracking system that keeps track of bugs for individual or groups of developers. In this section, we choose Bugzilla as the subject because it has been used by at least 1268 companies, organizations, and projects, among which there are some well-known large free software projects, such as *Mozilla*, *Linux kernel*, *Apache*, and *Eclipse*.

As Bugzilla has a large number of projects, we use the bug reports of only *Eclipse*[5)] and *Apache*[6)], because these two projects are well known large open source systems and are widely used in empirical research. For each project, we recorded its following data from Nov, 31st to Dec 15th, 2013: (1) the total number of bug reports, (2) the proportion of different resolutions, (3) the number of daily generated bug reports, and (4) the number of daily handled bug reports. These results are shown by Figures 2–7.

Figures 2 and 3 presents the total number of bug reports for *Apache* and *Eclipse*, respectively. As a bug report has eight possible status in its life-cycle, we use different patterns to distinguish the eight status. Until Dec 15th, 2013, *Apache* project has received more than 39300 bug reports totally, and *Eclipse* has received more than 422800. From Figures 2 and 3, for *Apache*, 4222 (*UNCONFIRMED+NEW+ASSIGNED +REOPEN-ED*) bug reports need to be further handled, which accounts for 10.72% of total number; for *Eclipse*, 63155 (*UNCONFIRMED+NEW+ASSIGNED+REOPENED*) bug reports need to be further handled, which accounts for 14.94% of total number. That is, developers have to handle a large number of bug reports.

4) http://www.bugzila.org.
5) http://www.eclipse.org.
6) http://www.apache.org.

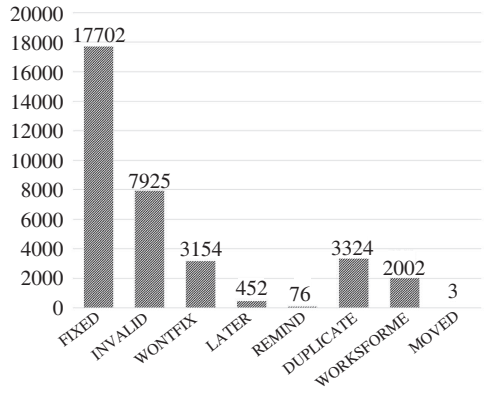

Figure 4 Number of resolutions for Apache.

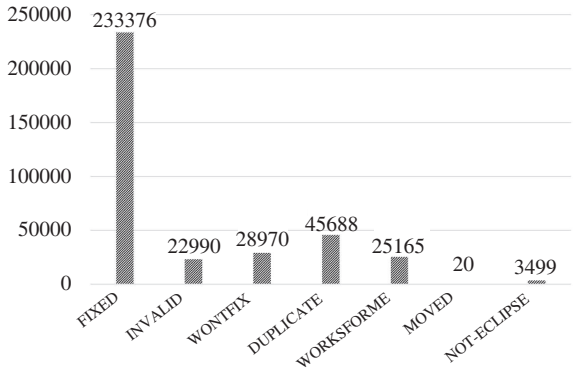

Figure 5 Number of resolutions for Eclipse.

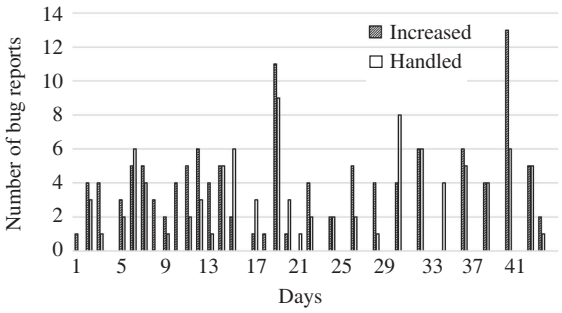

Figure 6 Number of bug reports submitted and handled every day for Apache.

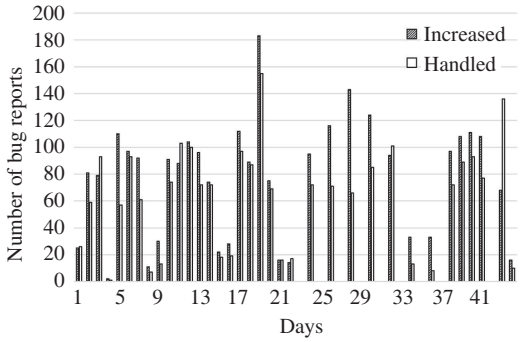

Figure 7 Number of bug reports submitted and handled every day for Eclipse.

Figures 4 and 5 presents the number of resolutions of bug reports for *Apache* and *Eclipse*, which show the quality of the reports. The horizonal axis depicts the possible resolutions of bug reports defined by *Bugzilla*. In particular, "Fixed" depicts that the bug reports actually record bugs and the recorded bugs have been fixed, whereas the other resolutions (e.g., "Invalid", "Duplicate", and "Later") depict the bug reports whose bugs have not been fixed due to some reasons (e.g., invalid bug reports, duplicate bug reports, or leaving the bug for later). The vertical axis depicts the total number of bug reports of the same resolution status. From Figures 4 and 5, of all the handled bug reports, 2224 (6.42%) bug reports of *Apache* and 45688 (12.70%) of *Eclipse* are *DUPLICATION*, whereas 7925 (22.88%) bug reports of *Apache* are *INVALID* and 2002 (5.78%) bug reports of *Eclipse* are *WORKSFORME* (i.e., developers cannot reproduce the bug). That is, the quality of bug reports is not good enough.

Figures 6 and 7 presents the number of bug reports submitted or handled every day for *Apache* and *Eclipse*, whose horizonal axis depicts each day from Nov, 31th to Dec 15th, 2013. From this figure, for *Apache*, the number of newly increased and handled bug reports is not so large. In particular, 2.8 bug reports were generated on average every day, and 2.2 bug reports were handled. For *Eclipse*, the number of newly increased and handled reports is larger. In particular, every day, 60 bug reports are generated and 50 bug reports are handled on average. That is, handling bug reports is usually as efficient as reporting bug reports. However, for *Eclipse*, sometimes the largest number of increased bug reports everyday is 180. That is, sometimes it is costly for developers to triage and fix bug reports due to the existence of a large number of bug reports.

As shown by Figures 4 and 5, the quality of bug reports is far from good, and thus, many researchers work on optimizing bug reports. Considering the large number of bug reports shown by Figures 2–5, it is tedious and costly for developers to triage bug reports and fix bugs reported by bug reports, and thus many researchers worked on automating bug-report triage and bug fixing.

Table 1 Classification scheme

Primary classification	Secondary classification	Bug-report status
4 Bug-report optimization	4.1 Content optimization of bug reports 4.2 Work on bug-report misclassification 4.3 Prediction on bug-report severity 4.4 Discussion	From *UNSOLVED* to *NEW*
5 Bug-report triage	5.1 Bug-report prioritization 5.2 Detection of duplicate bug reports 5.3 Bug-report assignment 5.4 Discussion	From *NEW* to *ASSIGNED*
6 Bug fixing	6.1 Bug localization based on bug reports 6.2 Recovering links 6.3 Bug-fixing time prediction 6.4 Discussion	From *ASSIGNED* to *RESOLVED*

3 Classification scheme

In our survey, we employ the classification scheme from two perspectives: the reporters' perspective and the developers' perspective. From the reporters' perspective, there are some research on improving the quality of bug reports. From the developer's perspective, there are many research on automating bug-report triage and fix. That is, we classify the existing research related to bug report into three categories: bug-report optimization (in Section 4) , bug-report triage (in Section 5), and bug fixing (in Section 6).

As Table 1 shows, on bug-report optimization, we will introduce the approaches that researchers adopted to improve the quality of bug reports, including optimizing bug-report (in Subsection 4.1), avoiding bug-report misclassification (in Subsection 4.2), and predicting bug-report severity (in Subsection 4.3); on bug-report triage, we will introduce the current techniques to automate the process of bug triage, including prioritizing bug reports (in Subsection 5.1), checking duplication of bug reports (in Subsection 5.2) and assigning bug reports to developers (in Subsection 5.3); on bug fixing, we will introduce the research on bug-report based bug fixing, including localizing bugs (in Subsection 6.1), recovering links between bug reports and the corresponding fixing changes (in Subsection 6.2), and predicting bug-fixing time (in Subsection 6.3). At the end of each of the three general categories, we will shortly discuss the existing problems as well as the possible future research directions.

4 Bug-report optimization

Bug reports play an important role in software development. As the quality of bug reports influences the time of bug fixing, bug-report optimization becomes important. However, in practice, many bug reports do not have high quality. For example, according to Subsection 2.3, of the total bug reports, 22.88% bug reports of *Apache* are invalid. To reduce invalid bug reports and improve the quality of bug reports, researchers investigated the following main ways to improve the quality of bug reports: (1) study on the content of bug reports, (2) techniques on reducing bug-report misclassification, and (3) techniques on predicting the severity of bug reports.

4.1 Content optimization of bug reports

As discussed in Section 2, a typical bug report may include much information, including the bug's severity or importance, the programmer assigned to fix the bug, its resolution status, steps to reproduce the bug, expected behavior, stack traces, proposed patches and test cases. However, due to the practical concern, it is tedious and impossible to include as much information as possible in the bug reports. Therefore, it

is necessary to investigate the essential information for a bug report and researchers usually studied this problem by performing some investigations.

Bettenburg et al. [5] first studied the quality of bug reports in 2007. They conducted a survey on some developers[7)] of *Eclipse* and gave a tool to measure the quality of bug reports according to their content. Their results showed that *steps to reproduce* and *stack traces* are considered most important by developers, and errors in *steps to reproduce* and incomplete information are most harmful. In the next year, Bettenburg et al. [4] expanded their survey to three projects (*Apache*, *Eclipse*, and *Mozilla*). They collected more data (i.e., 466 responses, including 156 from developers and 310 from reporters) and got more convincing results. For developers, the most widely used information of bug reports includes *steps to reproduce*, *expected behavior*, and *stack traces*, and the most important information they think includes *steps to reproduce*, *stack traces*, and *test cases*. For users, few of them added *stack traces* and *test cases* to their bug reports, though they consider *steps to reproduce* and *test cases* most helpful to developers. They also trained a new tool by building supervised learning models using the quality ratings of bug reports from developers. The tool not only measures the quality of a new bug report, but also suggests reporters how to enhance their bug reports. Hooimeijer and Weimer [6] also presented a model to measure bug-report quality, but they divided bug reports into "cheap" and "expensive" by predicting whether bug reports can be resolved within a given time. Their research was performed based on the assumption that "in general, reports of higher quality are dealt with more quickly than those of lower quality". This assumption, however, is not so reasonable, as many bug reports are addressed quickly not because of their quality, but because of the urgent problems they describe. Xie et al. [17] conducted an empirical study on Gnome and Mozilla to understand how the projects leverage non-developer volunteers (called triager) to improve the quality of bug reports. They found the primary impact of triager involves filtering reports, filling missing information and determining the relevant product. They pointed out that triagers were good at filtering invalid reports but had difficulty in accurately pinpointing the relevant product.

Furthermore, Breu et al. [18] performed a questionnaire on 600 bug reports from $Mozilla$ and *Eclipse*. From the questionnaire, the information needs change over the life-cycle of a bug report. Furthermore, similar to Bettenburg et al. [4], they suggested users to provide screenshots, stack traces, and steps to facilitate reproducing as well.

For the research on specific bug report fields, Lamkanfi et al. [19] noticed that bug reports may contain errors such as wrong components, so they proposed a data-mining technique to predict wrong components for a particular bug report. Herraiz et al. [20] held the opinion that the reports of Bugzilla are too complex, so they classified bug reports by the close time and by levels of severity, and found that there are more clusters for the latter division standard. They finally concluded that the severity of bug reports can be reduced from seven options to three.

In the view of the whole bug report, Wu et al. [21] mentioned that the bug reports are often incomplete, so they proposed BUGMINER, which derived useful information from bug-report databases, and used the information to check the completion of a given bug report. Xia et al. [22] mentioned that the contents of bug reports may be changed due to several reasons. To investigate the problem, they first asked some bug reporters and developers the reasons of changing the contents, and then conducted an empirical study on four open-source projects (i.e., OpenOffice, Netbeans, Eclipse, and Mozilla). From their study, fixing a changed bug report takes more time. Rastkar et al. [23] generated the summaries for bug reports as they thought that an accurate summary can help developers quickly get to the right bug report when they make use of existing bug reports for analysis.

Instead of focusing on the contents of bug reports, Ko et al. [24] performed a linguistic analysis of 200000 bug report titles from five projects (i.e., *Eclipse*, $Firefox$, *Apache*, *Linux*, and $OpenOffice$) to study how people describe software problems and suggested new designs for more structured report forms. In particular, they applied a probabilistic part-of-speech tagger [25] to all report titles, by counting the nouns, verbs, adverbs, and adjectives that appeared in those titles, and proposed several design ideas motivated by their results, such as soliciting more structured titles.

7) Actually, they asked 336 developers, but received only 48 responses.

4.2 Work on bug-report misclassification

In 2008, Antoniol et al. [26] raised the problem of bug-report misclassification, which is to distinguish bugs from non-bugs. Furthermore, they built three classifiers (i.e., alternating decision trees, Naive Bayes classifiers, and logistic regression) to distinguish bugs from other issues (e.g., enhancement and refactoring) on *Mozilla*, *Eclipse*, and *JBoss*, whose precision is from 77% to 82%. Pingclasai et al. [27] proposed another approach to automatically classifying bug reports—topic modeling. Also, they compared three classification models (i.e., decision tree, Naive Bayes classifier, and logistic regression) to find the most effective one. According to their results, Naive Bayes classifier is the most efficient classification model. Later, Herzig et al. [28] manually inspected more than 7000 issue reports from two tracking systems *Bugzilla* and *Jira* to learn the percentage of bug reports that have been misclassified and found that "Every third bug is not a bug". Besides, they found 40% issue reports are inaccurately classified, and 39% files that are marked as defective have never had a bug.

From a different respect, Zanetti et al. [29] conducted a case study on four open-source software communities, and found that bug reporters' centrality plays an important role in bug-report quality, based on which they developed an automated bug-report classification mechanism using a support vector machine, to predict whether a reported bug is valid or faulty.

4.3 Prediction on the severity of bug reports

"Severity" is an important content of a bug report assigned by reporters, which is critical for developers to decide the priority of different bugs. Reporters (e.g, testers or end-users) often fail to recognize the severity of bug reports due to the lack of experience or some other reasons.

To help testers assign severity levels to bug reports, in 2008, Menzies and Marcus [13] presented the first automated approach, which is based on standard text-mining and machine-learning techniques. Their severity predictor first used typical text-mining techniques (i.e., tokenization, removing stop words, and stemming) to deal with textual description of bug reports, then used a Java version of Cohen's RIPPER rule learner [30] to generate rule sets based on important tokens got before. They emphasized that severity prediction is especially important for mission critical systems (e.g., systems developed by NASA), and performed a case study using data from NASA's Project and Issue Tracking System (PITS). Similarly, Lamkanfi et al. [31] used the same technique (text mining and machine learning) to predict severity of bug reports by using larger training sets from three open-source projects: *Mozilla*, *Eclipse*, and *Gnome*. Different from the previous techniques, they used a Naive Bayes classifier to train the tokens. Their research goal is comparatively easy, which is to distinguish non-severe bugs from severe bugs. In particular, in their approach, *trivial* and *minor* of severity levels are taken as *non-severe*, whereas *major*, *critical*, and *blocker* of severity levels are taken as *severe*. Later, Lamkanfi et al. [32] completed a follow-up study, which is a comparison of four well-known document classification algorithms (i.e., Naive Bayes, Naive Bayes Multinomial, K-Nearest Neighbor, and Support Vector Machines) on bug-report severity prediction. According to this study, Naive Bayes is best suited for classifying bug reports into "severe" or "non-severe". Extending the above work, Tian et al. [33] proposed a new information retrieval based approach to predicting the severity levels of bug reports. In particular, they used BM25F similarity function to evaluate the similarities among different bug reports, then used the similarity in a nearest-neighbor fashion to assign severity levels to bug reports. Their approach uses both textual and non-textual information of bug reports. Their work is mostly similar to Menzies and Marcus's work [13] because both of them is fine-grained (i.e., five severity labels). However, the former work is evaluated to outperform Menzies and Marcus's work.

4.4 Discussion

It is necessary to improve the quality of bug reports due to the existing problems in practical bug reports. First, reporters still report issues in the same way as bugs by bug reports in some bug tracking systems, although these issues may be feature request, missing documentation, and so on. Second, it is hard to automatically distinguish real bugs from non-bugs, and thus developers usually have to manually select

real bugs. Third, some bug reports are invalid because reporters sometimes fail to report contents that are essential for bug fixing. Fourth, reporters can hardly submit bug reports when they cannot reproduce bugs.

Considering the techniques used in bug-report optimization, researchers usually use questionnaires to investigate what contents of a bug report are essential or important; when to distinguish bugs from other issues or predict bug-report severity, researchers usually use text mining and machine learning techniques.

In the future, to improve the quality of bug reports, bug tracking systems may check the quality of bug reports as reporters submit a bug report. For example, bug tracking system may check whether a bug report is duplicate as soon as reporters submit it so as to reduce the large number of duplicate bug reports before bug-report triage. Moreover, it is necessary to specify the necessary contents of bug reports before they are submitted so as to reduce the number of invalid bug reports that cannot be solved by developers.

5 Bug-report triage

Bug-report triage is a process of checking bugs, prioritizing bugs, and assigning bugs to proper developers (i.e., bug assignment) [17]. In practice, most bug reports are triaged manually to developers [34]. However, for a large open-source project, every day an ignorable number of bug reports are submitted. For example, 60 bug reports are generated every day on average for the open-source version of *Eclipse*. That is, it is a labor-intensive task for developers to read, analyze, prioritize these bug reports, check duplication of these bug reports, and assign these bug reports to a proper developer from hundreds of candidates. Moreover, these procedures are also error-prone [17,35].

To assist triagers who are responsible for the process of bug-report triage and improve the bug triaging efficiency, many researchers have focused on automating bug-report triage, including prioritizing bug reports (in Subsection 5.1), checking duplication of bug reports (in Subsection 5.2) and assigning bug reports to developers (in Subsection 5.3).

5.1 Bug-report prioritization

As a large number of bug reports exist in the bug tracking system for a large project, it is time-consuming and effort-consuming to deal with these bug reports. Some bug reports are more important than others. Due to time limit, it is necessary to deal with important bug reports early. That is, triagers assign priority to bug reports and prioritize these bug reports based on their priority.

To facilitate bug-report prioritization, Yu et al. [36] adopted neural network techniques. Also, they reused data sets from similar software systems to speed up evolutionary training, which aims to solve error problems. Kanwal and Maqbool [37] proposed a bug priority recommender based on SVM and Naive Bayes classifiers. Besides, they compared the results of these two classifiers in terms of accuracy, and found that when using text features for recommending bug priority, SVM performs better, while when using categorical features, Naive Bayers performs better. Tian et al. [38] presented a new machine learning framework and a new classification engine, which are used to predict the priority of bug reports considering multiple factors including *temporal*, *textual*, *author*, *related-report*, *severity*, and *product*. These factors are extracted as features and are then used to train a discriminative model via a classification algorithm.

5.2 Detection of duplicate bug reports

In 2005, Anvik et al. [39] conducted a large-scale empirical study on the bug repositories of *Eclipse* and $Firefox$, and found that a large portion (20%–40%) of bug reports are identified as duplicate by the developers. This study provided evidence for the existence of bug-report duplication and showed that it is necessary to detect duplicate bug reports and understand the phenomena of bug-report duplication.

The work on bug-report duplication can be classified into three categories. The first two categories focuses on presenting new techniques on detecting duplicate bug reports, whereas the last category focuses

on understanding bug-report duplication. In particular, the work of the first category mainly relies on the natural language information in bug reports, the work of the second category mainly relies on the execution information recorded by bug reports, and the work of the third category is mainly empirical studies on bug-report duplication.

5.2.1 *Natural-language based duplicate bug-report detection*

Hiew [40] made the first attempt to detect duplicate bug reports based on the textual information of bug reports. Specifically, his approach transforms the textual part of bug reports into word vectors, calculates the similarity between word vectors, and ranks candidate duplicate bug reports according to their similarity to a given bug. Runeson et al. [9] did another pioneer work on natural language based duplicate bug-report detection, in which they considered more textual features (e.g., software versions, testers, the submission date, and so on), and conducted a large-scale experimental study on industrial projects. Later, Jalbert and Weimer [41] proposed duplicate bug-report classification, in which they not only ranked a list of existing bug reports that are more similar with a new bug report, but also gave a recommendation on whether the new bug report is a duplicate bug report based on the similarities. Sureka and Jalote [42] proposed a novel approach to detecting duplicate bug reports based on N-grams of characters. Since their approach is not word-based, it is able to handle bug reports written in Asian languages such as Chinese and Japanese.

Furthermore, many works in this direction tried to enhance the accuracy of duplicate-bug-report detection based on the following ways: metrics on textual similarity, ranking strategies, or extra information. We summary and introduce existing work based on the three ways as follows.

(1) Improvement based on metrics for textual similarity. Sun et al. [43] proposed a novel approach based on discriminative approach that assigns different weights to words in bug reports when calculating textual similarity. Their empirical studies showed that their approach outperforms all former techniques that used merely textual information. Later, they further improved their approach [44] by bringing in a textual similarity metric called BM25F [45]. After that, also based on BM25F, Tian et al. [46] proposed to consider the similarity between new bug reports and multiple existing bug reports (instead of just the most similar bug reports) to decide whether the bug reported by the new bug report is actually a duplicate bug. Nyugen et al. [8] proposed to apply topic modelling to detect duplicate bug reports, which measures the semantic similarity between words. Different from the previous work that models a bug report as a set of words, Banerjee et al. [47] proposed to further consider word sequences when calculating the textual similarity between bug reports.

Furthermore, Falessi et al. [48] conducted a large-scale experimental study to compare a wide range of natural language processing techniques on the duplicate-bug-report detection in a large industrial project. They drew the conclusion that different NLP techniques do not have significant difference, but a proper combination of these techniques may result in a higher effectiveness.

(2) Improvement on ranking strategies. Liu et al. [7] proposed to apply a recently developed supervised machining learning technique called Learn To Rank (LTR) to rank candidate bug reports based on a series of features. Simultaneously, Zhou and Zhang [49] also proposed an LTR-based approach, using nine more sophisticatedly defined features.

(3) Improvement by using extra information. Feng et al. [50] proposed to use the profile information of the bug reporter to enhance the effectiveness of existing techniques on detecting duplicate bug reports. Alipour et al. [51] proposed to supplement existing approaches with domain knowledge (i.e., Android-specific keywords in their study), and their experiments showed that such domain knowledge can effectively enhance the accuracy of duplicate-bug-report detection.

5.2.2 *Execution information based duplicate bug report detection*

As many software platforms (e.g. Microsoft Windows, Firefox) provide features to record execution traces for field bugs, it is natural to use such execution information to detect duplicate bug reports because such information demonstrates the behavior of bugs. Therefore, many research efforts have been made on using execution information in duplicate-bug-report detection.

Podgurski et al. [14] started the preliminary work by proposing an approach to categorizing software failure reports by performing undirected clustering on their execution traces. Later, Wang et al. [52] proposed the first approach to detecting duplicate bug reports using execution traces. To learn the effectiveness of execution traces on detecting duplicate bug reports, they also combined execution traces with natural language information using a self-adapted weighing algorithm to assign weights to the information sources, and found that this combination technique achieves a higher accuracy than existing techniques using only natural language information. Based on this work, Song et al. [53] did some adaptations, and implemented a more efficient version for the IBM Jazz Project. Lerch et al. [54] proposed an approach to identifying stack traces in bug reports, transforming stack traces to a set of methods, and detecting duplicate bug reports by calculating and ranking the similarity between method sets. Recently, Kim et al. [55] proposed to use crash graphs to depict a number of related crash reports (stack traces), and detect duplicate crash reports by measuring the similarity between crash graphs. Dang et al. [56] proposed a novel model to measure the similarity between two stack traces. Specifically, in their work, they further considered the the distance of the matched functions to the top frame, and the offset distance between the matched functions.

5.2.3 *Empirical studies on duplicate bug reports*

Due to the existence of a large number of duplicate bug reports in software repositories, some researchers began to conduct empirical studies to better understand this phenomena. Specifically, Bettenburg et al. [57] conducted an empirical study on the influence of duplicate bug reports and found that although these bug reports cause extra burden on bug triaging, they also provide extra useful information that the original bug report may not cover. Wang et al. [58] studied existing duplicate bug reports, and found that many different technical terms actually have a similar meaning in the setting of software bugs. So they proposed an approach to detecting different software technical terms with similar meanings, based on known duplicate bug reports. Later, Cavalcanti et al. [59,60] conducted two empirical studies to find relations between the characteristics (e.g., size, life-time, bug-report submitters) of software repositories and bug-report duplication, and found that bug-report submitters with sporadic profiles are more likely to submit duplicate bug reports. Davidson et al. [61] conducted another empirical study to find the factors that affect bug-duplication rate, in which they found that software size and user groups do not have strong correlations with the bug-duplication rate.

5.3 Bug-report assignment

We classify and summary the existing work on bug-report assignment from two aspects, the techniques used in bug-report assignment and the information used in bug-report assignment.

5.3.1 *Classification based on technologies used in bug-report assignment*

After a comprehensive investigation, we found that work in bug-report assignment usually uses the following techniques, machine-learning techniques, information-retrieval techniques, and some mathematical theories like fuzzy sets and Euclidean distance. Therefore, we will review these work based on the techniques used in bug-report assignment:

(1) Machine learning based bug-report assignment. Most bug-report triage approaches use *Machine Learning* techniques for the purpose of text categorization. In particular, these approaches usually train a classifier with previously assigned bug reports and then use the classifier to classify and assign new bug reports.

Cubranic and Murphy [16] proposed the first machine-learning based bug-report triage approach in 2004. In particular, they viewed bug-report assignment as a *supervised learning* problem where the history information on developers and their fixed bugs (recorded in the bug reports) are used as training data to learn which developer a bug report should be assigned to. Based on their experimental study, the proposed approach based on a Naive Bayes classifier assigned 15859 bug reports and achieved 30% classification accuracy. Later, Anvik et al. [3,10] extended the previous work by filtering out noise data including bug

reports labeled "wontfix" or "worksforme", and developers who no longer worked there or contributed poorly. Compared with the previous work, this approach is evaluated to have higher precision (more than 50%). Besides, based on the experimental results, among the three machine-learning algorithms, Naive Bayes, Support Vector Machines (i.e., SVM), and C4.5, SVM based bug-report assignment is competitive. To improve the accuracy of bug-report triage, Bhattacharya and Neamtiu [62] proposed to use refined classification, richer feature vectors and tossing graphs. From their experimental results, intra-fold updates are useful for improving the prediction accuracy in bug-report triage. Most recently, Hao et al. [63] proposed a novel recommendation method for large projects called BugFixer. They developed a new tokenization algorithm and used Vector Space Model (i.e., VSM) to compute bug report similarity. Their evaluation shows that BugFixer outperforms previous recommendation methods based on Naive Bayes and SVM.

Most of the related work on bug-report assignment is based on free and open-source projects, but Lin et al. [64] conducted an empirical study on a Chinese proprietary software project. Also using SVM, they conducted an experiment using Chinese text in bug reports (i.e., *title*, *description*, and *step*). Besides, they conducted another experiment based on non-text fields of bug reports (i.e., BUGTYPE, BUGCLASS, PHASEID, SUBMITTER, MODULEID, BUGPRIORITY) using J48 classifier and decision tree, and found that text data are more useful than non-text data when automatically assigning bug reports.

Unlike other supervised or unsupervised learning approaches on bug-report assignment, Xuan et al. [65] first proposed a semi-supervised approach. They proposed a semi-supervised learning approach by combining a Naive Bayes classifier and expectation-maximization, which generates a weighted recommendation list for bug-report triage. They took advantage of both labeled and unlabeled bug reports.

Different from the preceding work that focuses on finding the best developers, Alenezi and Magel [66] built a classification model to redistribute the load of overloaded developers.

Feature selection is an important component of machine learning, which may facilitate training-set reduction. Zou et al. [67] first combined feature selection (i.e., CHI Feature Selection Algorithm) with instance selection (i.e., ICF Instance Selection Algorithm) to improve the accuracy of bug-report triage and reduce the training sets. From their experimental results, their approach removed 70% words and 50% bug reports in applying machine learning to assign bug reports, and the reduced training sets provide better accuracy. Park et al. [35] reformulated bug assignment as an optimization problem of both accuracy and cost, and proposed a cost-aware triage algorithm, which extracts bug features to train an SVM model and is evaluated to have reduced 30% of cost.

(2) Information retrieval based bug-report assignment. Information Retrieval is also widely used in bug-report assignment, because bug reports are documents recording the information that may be used in bug-report assignment.

Following the hypothesis that "given a new change request, developers that have resolved similar change requests in the past are the best candidates to resolve the new one", Canfora and Cerulo [68] used the textual description of change request (i.e., bugs or features) as a query to find candidate developers by using information retrieval and thus constructed a competence database of developers. However, for a bug report, "most experienced developers" may not be "the best developers", their proposed approach may ignore some developers who have contributed in some code that is very related to this bug report because these developers have never dealt with a bug report before.

Term selection is a group of information-retrieval techniques which help to improve the accuracy of bug-report assignment. Alenezi and Magel [66] investigated the use of five term-selection techniques on the accuracy of bug-report assignment. Matter et al. [69] proposed a novel expertise model of developers based on the vocabulary found in the source code contributions of developers. Then they extracted information from bug reports, looked it up in the vocabulary, and recommended the top-k developers. They used information retrieval techniques to match the word frequencies in bug reports to the word frequencies in source code. Unlike other approaches, their approach does not rely on the quality of previous bug reports and can assign bug reports to a proper developer although he/she never worked on a bug report previously.

(3) Tossing graph. Jeong et al. [34] introduced a *Tossing Graph* model to reduce bug tossing (i.e., the process of reassigning a bug to other developers [34]). The proposed model is effective as it has been evaluated to reduce 72% of bug tossing events and improve 23% of automatic bug-report assignment accuracy. Furthermore, Bhattacharya and Neamtiu [62] extended their work by using additional attributes on edges and nodes, which has been evaluated to reduce 86.31% of tossing paths and achieve 83.62% of prediction accuracy in bug-report assignment.

(4) Fuzzy set. Tamrawi et al. [11] used a *Fuzzy Set* to represent the developers who have the bug-fixing expertise relevant to a specific technical term, and "determine the capable developers for the technical aspects in the system based on their past fixing activities". They proposed a tool Bugzie that combines fuzzy sets with tossing graphs, which has been evaluated to achieve higher accuracy and efficiency than other work.

(5) Euclidean distance. Xia et al. [70] extended bug fixer recommendation to bug resolution recommendation. In particular, they performed two kinds of analysis: bug reports based analysis and developer based analysis. In bug reports based analysis, they recommended developers according to similar bugs resolved in bug reports history. In developer based analysis, they measured the distance between a potential developer and a bug report by considering bug reports the developer resolved before.

5.3.2 *Classification based on information used in bug-report assignment*

In the research on bug-report assignment, researchers usually build expertise models based on features (e.g., words or terms) extracted from different data sets, such as bug reports or other documents. For the original data sets, most researchers prefer to assign new bug reports by using previously assigned bug reports [3,11,16,62,64,65,67,71]. Cubranic and Murphy [16] first used the *summary and description* of each report; Anvik et al. [3,10] removed unconfirmed or reopened reports and reports assigned to resigned or unexperienced developers; Baysal et al. [71] used the summary, the description, and *the comments*; Bhattacharya et al. [62] and Tamrawi et al. [11] used *the bug report's ID, the fixing developer's ID*, and summary as well as description; Park et al. [35] extracted features from the text description of bug reports and its *metadata* (i.e., version, platform, and target milestone). Besides, Baysal et al. [71], Zou et al. [67], Tamrawi et al. [11], Aljarah et al. [72] and Alenezi et al. [66] all used reduced terms to improve the accuracy and efficiency of bug-report assignment.

Some approaches are based on the history of **source code**. Matter et al. [69] proposed a bug-report triage approach based on the vocabulary found in the source code contributions of developers; Servant and Jones [73] combined the source-code history with the diagnosis information about the location of faults; Shokripour et al. [74] proposed an approach to predicting what source code files will be changed when fixing a new bug report based on its identifier, commit message, comments in source code and previously fixed bug reports.

As no prior work combined bug reports with source code, Kevic et al. [75] first proposed a collaborative approach, which lists similar bug reports though information retrieval and then analyzes associated change sets and the developers who authored the change sets. However, their approach is evaluated based on only the authors' own projects, and thus the effectiveness of the proposed approach needs to be further evaluated.

5.3.3 *Empirical studies on bug-report reassignment*

As developers may not deal with the bug reports assigned to them, sometimes bug-report triage has to reassign bug reports. Therefore, besides the work on automatic bug-report assignment, there is also some research on the analysis on bug-report reassignment. Guo et al. [76] conducted a large-scale study on the bug-report reassignment process of Microsoft Windows Vista to learn the reasons for bug-report reassignment. In particular, this study categorized five reasons for reassignments (i.e., finding the root cause, determining ownership, poor bug report quality, hard to determine proper fix, and workload balancing). Xie el al. [77] studied Mozilla project and found that the chance of bug report being reassigned is associated with the assigner's past activities and her/his social network. To prevent developers from

being distracted by reassignment and to crowd-source non-developers to improve reassignment, they proposed a logistic regression based tool to estimate the odds that the assignment is incorrect. Instead of studying bug-report assigner reassignment, Xia et al. [22] conducted an empirical study on bug report field reassignment, including reassigning reports to new developers. From their study, it usually takes more time to fix a reassigned bug than a non-reassigned bug.

5.4 Discussion

In summary, many researchers have focused on automated realization of bug-report triage. Due to various reasons, none of the existing approach has achieved satisfactory accuracy (e.g., more than 95%). Due to the accuracy concern, it is hard to apply existing automatic bug-report triage approaches in practice. Moreover, existing bug tracking systems are not able to provide any automatical support for bug-report assignment. Therefore, developers still detect duplication and assign bug reports manually.

The main techniques used in bug-reports triage are information retrieval and machine learning. Specially, information retrieval is used to split a bug report into words which are easy to inquire, and machine learning is used for the purpose of text categorization. Other techniques like tossing graph and fuzzy set based techniques are not generally used.

In the future, researchers on bug-report triage may focus on improving the accuracy of bug-report triage by using some new techniques (e.g., deep learning techniques) and new data sources (e.g., emails, comments of bug reports, documentation, logs).

6 Bug fixing

Besides the research in bug-report optimization and bug-report triage, there is also some research on bug-report based bug fixing, including localizing bugs and recovering links between bug reports and the corresponding fixing changes.

6.1 Bug localization based on bug reports

As soon as a bug is identified, the developer should locate the source code files (or methods, classes, etc.) that contain the reported bug [78]. The process of finding the location of bugs is called *bug localization*. Manual bug localization is time-consuming and tedious, especially for a large project with thousands of files. To reduce the bug-fixing time and maintenance cost, many researchers have proposed automated bug-localization techniques. In this section, we focus on only the techniques related to bug reports. Based on the information used in the techniques, we divide existing work on bug report based bug localization into two categories, bug localization based on source code and bug localization based on bug localization history.

6.1.1 *Source code based techniques*

Given a bug report for a new bug, source code based bug-localization approaches typically use information retrieval to rank the source code files by their relevance to a bug report. Similar to the operation process of a search engine, the preceding process takes a bug report as a query in information retrieval. According to Gay et al. [79] and Zhou et al. [80], the process of such an information-retrieval based bug localization consists of four steps, corpus creation, indexing, query formulation, and retrieval and ranking.

Corpus creation. This step performs lexical analysis for each source code file and creates a vector of lexical tokens. Some text transformation techniques are used, such as tokenizing, stopping, and stemming. To reduce the size of corpus, existing techniques usually remove stop-words (e.g., "the", "of", "to", and "for") that help form sentence structures but contribute little to the description of the topics, keywords (e.g., int, double, char, etc.), separators, and operators, since they usually have no impact on bug-localization effectiveness. To increase the likelihood that words in bug reports and source code files will match, words derived from a common stem are grouped, and replaced with one designated group

member (e.g., replace "fishes" and "fishing" with "fish"); composite tokens are also split into individual tokens (e.g., the variable "TypeDeclaration" is split into "type" and "declaration").

Indexing. This step indexes all the files in the created corpus. Inverted indexes are by far the most common form, and for every index term (word, stem, phrase, etc.), a list of documents that contain the index term is created. By using these indexes, fast query processing is enabled.

Query formulation. In this step, a bug report is treated as a query, and its textual description is processed with the same text transformation techniques (e.g., extract tokens, remove common words, stem each word, etc.) used in source code files. Index terms that are comparable to the source code terms are produced, and are used to search for relevant files in the indexed corpus.

Retrieval and ranking. In this step, files in the indexed corpus are ranked based on their textual similarity with transformed bug reports. A ranking algorithm based on retrieval model can calculate scores of these files.

As the retrieval model impacts the effectiveness of ranking, various retrieval models and methods on deriving ranking algorithms have been proposed. The widely used models are as follows:

• Latent Semantic Indexing (LSI). Latent semantic indexing (also called latent semantic analysis) [81] is a technique on analyzing relationships between documents and terms based on Singular Value Decomposition (SVD). This technique assumes that related words occur in similar pieces of text and uses a term-document matrix to describe the occurrences of terms in documents. Marcus et al. [82] first used information retrieval method for concept location, where LSI was used to find semantic similarities between change request and modules of software and whose result is a list of source code files ranked by their relevance. Comparing with other static code analysis based techniques, they discovered that the proposed LSI based technique is independent of programming languages and requires only simple source code preprocessing. The effectiveness of the proposed approach is evaluated based on NCSA Mosaic web browser. After that they analyzed three static concept location techniques (i.e., regular expression matching, static program dependencies, and information retrieval) to see the strengths and weaknesses of the proposed techniques [83]. For the IR-based technique, they used the LSI [82]. Again in 2006, Poshyvanyk and Marcus [84] combined Formal Concept Analysis (FCA) and LSI to promote the problem of concept location. The results showed that the combined technique performed better than a single technique. After that, Poshyvanyk et al. [85] then proposed a feature location method called PROMESIR, which combined LSI and a probabilistic ranking technique (scenario-based probabilistic ranking). They chose LSI instead of other models as LSI "has been shown to address the problems of polysemy and synonymy quite well". Liu et al. [86] combined dynamic program analysis with static information retrieval technique. In particular, they used LSI to index the comments and identifiers from source code. Most of the proceeding work has shown that combining several techniques together may improve bug-localization performance.

• Latent Dirichlet Allocation (LDA). Latent Dirichlet Allocation is a generative three-level hierarchical Bayesian model, in which each document is viewed as a mixture of various topics that spit out words with certain probabilities. Lukins et al. [87] first applied LDA model to automate bug localization, and found that LDA can be successfully applied for bug localization. They also compared their results to similar proceeding studies on LSI, and found that LDA performed better than LSI. After that, Lukins et al. [88] extended their work by presenting five case studies to evaluate the accuracy and scalability of LDA-based bug-localization techniques. They found that LDA is widely applicable as it is suitable for open-source software of varying size and stability. Nguyen et al. [12] also extended LDA for bug localization by proposing BugScout, which correlates bug reports and source code files via their shared topics (i.e., the same technical aspects of the system).

• Vector Space Model (VSM). In vector space model, both documents and queries are represented as $t - dimensional$ vectors, each dimension corresponds to a separate index term, and each scalar of a dimension represents the weight[8)] of the index term. The *cosine* of the angle between a document vector and a query vector is used to measure the similarity between the document and the query [89]. In

8) The weight of a term in a document or query denotes the importance of the term, which is usually determined by the retrieval model.

information retrieval, VSM model is simpler than other models when representing a document. Gay et al. [79] proposed IRRF (Information Retrieval with Relevance Feedback) based concept location. They chose VSM as the information retrieval model, as at that time there was no clear winner among LSI, VSM, or LDA. Zhou et al. [80] proposed an approach BugLocator based on revised Vector Space Model, which differs from classic Vector Space Model in that it considers document length and similar bugs that have been resolved before. According to their experiments, BugLocator outperforms existing bug-localization methods.

Several research has focused on the comparison of these models. Rao and Kak [90] compared five genetic models (i.e., the Unigram Model (UM), the Vector Space Model (VSM), the Latent Semantic Analysis Model (LSA), the Latent Dirichlet Allocation Model (LDA), and the Cluster Based Document Model (CBDM).) as retrieval tools for the purpose of bug localization, and found that UM and VSM are more effective on retrieving relevant files than sophisticated models like LDA and LSA. Chawla et al. [91] compared VSM and LSI, but their results showed that in most cases, LSI performs better than VSM.

Besides the models above, Saha et al. [92] presented BLUiR, which uses a baseline "TF.IDF model". They believe that code constructs (i.e., class and method names) improve the accuracy of bug localization. Unlike traditional research during source code parsing and term indexing, their approach excludes language keywords (e.g., if, else, etc.) as they are also important identifiers. Their approach also indexes full identifiers and splits tokens. Kim et al. [93] proposed a two-phase recommendation model that eliminates inadequate bug reports in the first phase. Their model outperforms the one-phase model and Nguyen's BugCount [12]. Most recently, Ye et al. [94] ranked source code files by leveraging domain knowledge (e.g., API specifications and the syntactic structure of code) to help bug localization. Their approach can locate 70% of the bug reports with the top 10 recommendations. Wong et al. [95] found that the existing approaches can not deal with large files or stack traces. To deal with these two issues, they proposed segmentation and stack-trace analysis techniques to improve the performance of bug localization. Their approach is evaluated to significantly improve a representative bug localization approach.

6.1.2 *History based techniques*

Another typical way to localize a bug is to search for similar bugs resolved in the past and find a potential resolution, as the same bug may have been encountered and fixed in another code branch [96]. Cubranic et al. [97] built a tool to provide developers efficient and effective access to the group memory (i.e., the collection of software artifacts that contain a work group's past experience). The tool is able to provide a highly ranked recommendation that points out the relevant location and constructs. Ashok et al. [96] proposed a recommender system for debugging, which can automatically search through diverse data repositories for similar bugs from the past. According to their evaluation, 78% of the 129 users stated the recommendations were useful and 75% of them believed the search results useful. Zhou et al. [80] examined similar bugs resolved before to improve bug-localization performance.

Furthermore, Davies et al. [98] proposed an approach to combining source code files and past similar bugs. Their approach has been evaluated to be effective.

6.2 Recovering links between bug reports and change files

Bug reports are usually stored in bug-tracking systems. When bug reports are fixed, the corresponding changes of source code will be recorded in change logs. The links between bug reports and committed change logs are important in defect prediction, evolution analysis, and software quality measurement. However, such links are often missing due to many reasons (e.g., systems provide insufficient supports or developers forget to record such links).

To address this problem, many researchers focus on recovering the links between bug reports and change files. As some link recovery approaches yield bias results, some research focuses on revealing the bias and observing its negative impacts. In the following two subsections, we will review the existing work on these two aspects.

6.2.1 *Link recovery*

Change logs are usually stored in version archives, such as CVS. Fischer et al. [99] realized that there are insufficient supports for software evolution analysis between version control and bug tracking systems, so they populated a Release History Database, which stores extracted data. In particular, they combined bug reports with change logs by comparing bug report ID. That is, they used a table to store the bug report ID found in change logs. Furthermore, Fischer et al. [100], improved their previous approach by considering both ID and the names of bug reports in analyzing software features.

Sliwerski et al. [101] linked bugs with changes from two levels: syntactic level and semantic level. In the syntactic level, they split change logs into tokens and checked if there exists a bug number or key words such as "fixed" and "bug"; in the semantic level, they validated the link by checking whether the bug has been fixed, whether the change log contains the bug report description, and so on. Schroter et al. [102] applied this approach in analyzing where bugs come from. Zimmermann et al. [103] used a similar approach to predict defects for Eclipse.

Traditional link recovery approaches are effective, but suffer from poor accuracy. To improve the accuracy, Wu et al. [104] developed an automatic link recovery algorithm—ReLink, which first discovered links using traditional approaches, and then analyzed the missing links based on three features including time interval, bug owner and change committer, and text similarity. The ReLink yields higher accuracy and perform significantly better in maintainability and defect prediction. However, Bissyande et al. [105] pointed out that the evaluation of ReLink has several flaws, so they used more reliable data to evaluate ReLink, and found that ReLink is less effective in recovering missing links than recovering links that are actually correct.

All the proceeding approaches are based on textual similarity between bug reports and change files. Nguyen et al. [106] proposed a muti-layered recovery approach—MLink, which takes advantage of both textual features and code features. Their evaluation shows that MLink outperforms the state-of-the-art methods by improving 18% accuracy.

Despite these automatic methods in link recovery, Bird et al. [107] developed a tool, LINKSTER, to provide convenience for manually establishing links. This tool integrates bug reports, source code repositories, and mailing lists, and facilitates the exploration and annotation of software-engineering data.

6.2.2 *Bias revealing in link recovery*

Since the links between bug reports and change files are widely used in defect prediction, evolution analysis, and so on, it is essential to guarantee the accuracy and sufficiency of recovered links. However, several evaluation shows that linkage bias is inevitable.

Bird et al. [108] noticed that only a fraction of bug fixes are labeled in change files. They found bug feature bias and commit feature bias, and also provided strong statistical evidence on bug feature bias. Furthermore, they illustrated the potential adverse effect of bias, and claimed that "bias is a critical problem". Bachmann et al. [109] extended this work by doing a more thorough evaluation and presented a detailed examination of the bias in automatically recovered linked set. After that, Nguyen et al. [110] conducted a pioneer study using near-ideal data-set, but bias still exists, so they believe bias is not a result of datasets and approaches, but a symptom of software development process.

6.3 Bug-fixing time prediction

Predicting the time to fix a bug is important in maintaining software quality or aiding project planning process. Several techniques have been proposed to measure or predict bug-fixing time.

Most of the bug-fixing time prediction approaches are building prediction models based on the attributes of bug reports. Panjer [111] used the Bugzilla database of Eclipse for analysis. The attributes they used in modeling are *priority*, *severity*, *version*, *comments*, *components*, and so on. They finally achieved a prediction accuracy of 34.9% , and found that *comments*, *severity*, *products*, *component*, and *version* influence bug-fixing time the most. Giger et al. [112] combined the attributes of the initial bug

report with post-submission information (i.e., the information submitted after a bug was reported) to classify bug reports into "Fast" and "Slow" (which denotes the fixing time) and their approach has been evaluated to achieve an accuracy between 60% and 70%. Moreover, post-submission data was proved to improve the performance by 5% to 10%. Besides the attributes Panjer used, they also took account into other attributes, such as *assignee*. Most recently, Zhang et al. [113] performed a dedicated empirical study on bug-fixing time with three commercial projects of CA Technologies (i.e., a multinational company providing IT management software and solutions). Their results show that the Submitter and the Owner are the top 2 most important attributes which contribute to bug-fixing time prediction. Besides, they proposed methods both on predicting the number of bugs to be fixed and on estimating the time required to fix these bugs.

Instead of using the attributes of bug reports, Weiss et al. [114] used text similarity to predict bug-fixing time, which is to find similar bug reports resolved before and then provides those similar bug reports' bug-fixing time for consultation when facing a new bug report. Hooimeijer and Weimer [6] viewed time as an important measurement of bug-report quality, and measured bug-report-triage time instead of bug-fixing time using linear regression analysis.

Furthermore, some research takes developers who fix the bug reports as an important factor that may affect bug-fixing time. In particular, Anbalagan and Vouk [115] studied on more than 70000 bug reports from nine releases of Ubuntu, and found that "there is a strong linear relationship between the number of users participating in a bug report and the median time taken to correct it". Based on this finding, they proposed a linear model to predict bug-fixing time. Guo et al. [116] performed an empirical study on Windows Vista and Windows 7, and found that the higher reputation a developer has, the more likely his/her bug reports get fixed.

To evaluate the significance of different features used by previous prediction models (e.g., bug reports attributes, post-submission information, number of developers, and reporter's reputation), Bhattacharya and Neamtiu [117] conducted a large scale experiment on more than 500000 bug reports from five open-source projects, using both multivariate and univariate regression testing. Their findings are different from existing work. For example, they found no linear relationship between a reporter's reputation and his/her bug-fixing time.

Furthermore, as bug-report databases contain some long-lived bug reports that are not resolved due to some reasons, Saha et al. [118] analyzed these long-lived bugs from five aspects: proportion, severity, longest bug-fix process, reasons, and nature. Their study has the following findings: 5%–9% bugs take more than a year to be fixed. Bug assignment and bug fix are actually time-consuming. Moreover, the existence of long-lived bugs results from problem complexity, reproducing errors, and misunderstanding of bug-report severity.

6.4 Discussion

To conclude, the same as bug-report triage, none of the existing automated-fixing approach has achieved satisfactory accuracy, and developers still fix bugs manually. Information retrieval is also widely used in bug localization and links recovering.

In future, more techniques (e.g., deep learning) can be used in automating bug fixing, especially in bug localization based on bug reports. Also, to increase the accuracy of bug localization, it is essential to assure that reporters provide high-quality bug reports. On the other hand, some skills can be used for improving bug localization, such as adding some kind of labels in both the source code and the bug report.

7 Other work on bug-report analysis

Besides the preceding research on bug-report optimization, bug-report triage, and bug fixing, there is also a little research in other aspects of bug-report analysis that cannot be categorized into the above three scopes, with various motivations. We will introduce these research in this section.

As many developers use bug-tracking systems to discuss some issues in software development, Ko and Chilana [119] used the whole set of closed bug reports from three open-source projets (i.e., $Firefox$, *Linux kernal*, and $FacebookAPI$) to perform a qualitative analysis of design discussions in those reports. They found that many arguments in these discussions are about whether to realize original design intents or to make adaption according to user needs. They also suggested online discussion tools be redesigned for clearer proposals. Sahoo et al. [120] examined reproducibility of bug reports via randomly selecting bug reports of six server applications. They found that one request is enough for reproducing bugs in 77% cases. Xuan et al. [121] presented the first model to prioritize developers in bug-tracking systems using a socio-technical approach. In particular, by examining tasks in bug-report repositories, they concluded that developer prioritization is helpful to enhance bug-report handling, especially for bug-report triage. Bhattacharya et al. [122] defined several metrics to measure the quality of Android bug reports. On the other hand, they compared Google Code's bug tracker with Bugzilla and Jira, and found that though Google Code's bug tracker is more widely used by Android apps, it offers less management support. Wang et al. [123] built a state transition model based on historical data, and presented a method for predicting bug numbers at each state of a bug report, which can be used to predict a project's future bug-fixing performance.

8 Conclusion

There is considerable volume of research on bug-report analysis. After introducing some preliminaries, we presented some statistics on practical bug reports to show that ensuring bug reports quality, automating bug reports triage and localization are indeed urgent to be solved. Then we conducted a rather thorough survey on existing bug-report analysis work, mainly including bug-report optimization, bug-report triage, and bug fixing. Among the work in bug-report analysis, machine learning and information retrieval are main techniques widely used.

However, many problems are still unsolved, even have not been studied. Many researchers have focused on automated realization of handling bug reports (e.g., bug-report assignment, duplication detection, bug localization). Unfortunately, none of the existing approaches has achieved satisfactory accuracy (e.g., more than 95%). That is, it is hard to apply existing automatic approaches in practice. Furthermore, bug-tracking systems are still not able to provide any support for automatically dealing with bug reports. In the future, researchers may consider how to improve the accuracy of existing automatic approaches.

Acknowledgements

This work was supported by National Basic Research Program of China (973 Program) (Grant No. 2015CB3522-01), National High-tech R&D Program of China (863 Program) (Grant No. 2013AA01A605), National Natural Science Foundation of China (Grant Nos. 61228203, 61272157), and Fund for the Advisors of Beijing Excellent Doctoral Dissertations (Grant No. 20131000111).

References

1 Si X, Hu C, Zhou Z. Fault prediction model based on evidential reasoning approach. Sci China Inf Sci, 2010, 53: 2032–2046
2 Xie T, Zhang L, Xiao X, et al. Cooperative software testing and analysis: advances and challenges. J Comput Sci Technol, 2014, 29: 713–723
3 Anvik J, Hiew L, Murphy G C. Who should fix this bug? In: Proceedings of the International Conference on Software Engineering, Shanghai, 2006. 361–370
4 Bettenburg N, Just S, Schröter A, et al. What makes a good bug report? In: Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering, Atlanta, 2008. 308–318
5 Bettenburg N, Just S, Schröter A, et al. Quality of bug reports in Eclipse. In: Proceedings of the OOPSLA workshop on Eclipse Technology eXchange, Montreal, 2007. 21–25
6 Hooimeijer P, Weimer W. Modeling bug report quality. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, Atlanta, 2007. 34–43

7 Liu K, Tan H B K, Chandramohan M. Has this bug been reported? In: Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Cary, 2012. 28

8 Nguyen A T, Nguyen T T, Nguyen T N, et al. Duplicate bug report detection with a combination of information retrieval and topic modeling. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, Essen, 2012. 70–79

9 Runeson P, Alexandersson M, Nyholm O. Detection of duplicate defect reports using natural language processing. In: Proceedings of the International Conference on Software Engineering, Minneapolis, 2007. 499–510

10 Anvik J. Automating bug report assignment. In: Proceedings of the International Conference on Software Engineering, Shanghai, 2006. 937–940

11 Tamrawi A, Nguyen T T, Al-Kofahi J, et al. Fuzzy set-based automatic bug triaging. In: Proceedings of the International Conference on Software Engineering, Waikiki, 2011. 884–887

12 Nguyen A T, Nguyen T T, Al-Kofahi J, et al. A topic-based approach for narrowing the search space of buggy files from a bug report. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, Lawrence, 2011. 263–272

13 Menzies T, Marcus A. Automated severity assessment of software defect reports. In: Proceedings of the IEEE International Conference on Software Maintenance, Beijing, 2008. 346–355

14 Podgurski A, Leon D, Francis P, et al. Automated support for classifying software failure reports. In: Proceedings of the International Conference on Software Engineering, Portland, 2003. 465–475

15 Raymond E. The cathedral and the bazaar. Knowl Technol Policy, 1999, 12: 23–49

16 Čubranić D. Automatic bug triage using text categorization. In: Proceedings of the International Conference on Software Engineering & Knowledge Engineering, Alberta, 2004. 92–97

17 Xie J, Zhou M, Mockus A. Impact of triage: a study of mozilla and gnome. In: Proceedings of the ACM / IEEE International Symposium on Empirical Software Engineering and Measurement, Baltimore, 2013. 247–250

18 Breu S, Premraj R, Sillito J, et al. Information needs in bug reports: improving cooperation between developers and users. In: Proceedings of the ACM Conference on Computer Supported Cooperative Work, Savannah, 2010. 301–310

19 Lamkanfi A, Demeyer S. Predicting reassignments of bug reports-an exploratory investigation. In: Proceedings of the European Conference on Software Maintenance and Reengineering, Genova, 2013. 327–330

20 Herraiz I, German D M, Gonzalez-Barahona J M, et al. Towards a simplification of the bug report form in Eclipse. In: Proceedings of the International Working Conference on Mining Software Repositories, Leipzig, 2008. 145–148

21 Wu L L, Xie B, Kaiser G E, et al. BugMiner: software reliability analysis via data mining of bug reports. In: Proceedings of the International Conference on Software Engineering & Knowledge Engineering, Miami Beach, 2011. 95–100

22 Xia X, Lo D, Wen M, et al. An empirical study of bug report field reassignment. In: Proceedings of the Software Evolution Week—IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering, Antwerp, 2014. 174–183

23 Rastkar S, Murphy G C, Murray G. Summarizing software artifacts: a case study of bug reports. In: Proceedings of the ACM/IEEE International Conference on Software Engineering, Cape Town, 2010. 505–514

24 Ko A J, Myers B A, Chau D H. A linguistic analysis of how people describe software problems. In: Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing, Brighton, 2006. 127–134

25 Toutanova K, Klein D, Manning C D, et al. Feature-rich part-of-speech tagging with a cyclic dependency network. In: Proceedings of the Human Language Technology Conference of the North American Chapter of the Association for Computational Linguistics, Edmonton, 2003. 173–180

26 Antoniol G, Ayari K, Di Penta M, et al. Is it a bug or an enhancement? A text-based approach to classify change requests. In: Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research, Richmond Hill, 2008. 23

27 Pingclasai N, Hata H, Matsumoto Ki. Classifying bug reports to bugs and other requests using topic modeling. In: Proceedings of the Asia-Pacific Software Engineering Conference, Ratchathewi, 2013. 13–18

28 Herzig K, Just S, Zeller A. It's not a bug, it's a feature: how misclassification impacts bug prediction. In: Proceedings of the ACM/IEEE International Conference on Software Engineering, San Francisco, 2013. 392–401

29 Serrano Zanetti M, Scholtes I, Tessone C J, et al. Categorizing bugs with social networks: a case study on four open source software communities. In: Proceedings of the ACM/IEEE International Conference on Software Engineering, San Francisco, 2013. 1032–1041

30 William C. Fast effective rule induction. In: Proceedings of the International Conference on Machine Learning, Tahoe City, 1995. 115–123

31 Lamkanfi A, Demeyer S, Giger E, et al. Predicting the severity of a reported bug. In: Proceedings of the International Working Conference on Mining Software Repositories, Cape Town, 2010. 1–10

32 Lamkanfi A, Demeyer S, Soetens Q D, et al. Comparing mining algorithms for predicting the severity of a reported bug. In: Proceedings of the European Conference on Software Maintenance and Reengineering, Oldenburg, 2011. 249–258

33 Tian Y, Lo D, Sun C. Information retrieval based nearest neighbor classification for fine-grained bug severity predic-

tion. In: Proceedings of the Working Conference on Reverse Engineering, Kingston, 2012. 215–224
34 Jeong G, Kim S, Zimmermann T. Improving bug triage with bug tossing graphs. In: Proceedings of the joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, Amsterdam, 2009. 111–120
35 Park J W, Lee M W, Kim J, et al. Costriage: a cost-aware triage algorithm for bug reporting systems. In: Proceedings of the Conference on Artificial Intelligence, San Francisco, 2011. 139–144
36 Yu L, Tsai W T, Zhao W, et al. Predicting defect priority based on neural networks. In: Proceedings of the International Conference on Advanced Data Mining and Applications, Chongqing, 2010. 356–367
37 Kanwal J, Maqbool O. Bug prioritization to facilitate bug report triage. J Comput Sci Technol, 2012, 27: 397–412
38 Tian Y, Lo D, Sun C. DRONE: predicting priority of reported bugs by multi-factor analysis. In: Proceedings of the IEEE International Conference on Software Maintenance, Eindhoven, 2013. 200–209
39 Anvik J, Hiew L, Murphy G C. Coping with an open bug repository. In: Proceedings of the OOPSLA Workshop on Eclipse Technology Exchange, San Diego, 2005. 35–39
40 Hiew L. Assisted detection of duplicate bug reports. Dissertation for the Master Degree. Vancouver: The University of British Columbia, 2006
41 Jalbert N, Weimer W. Automated duplicate detection for bug tracking systems. In: Proceedings of the Annual IEEE/IFIP International Conference on Dependable Systems and Networks, Anchorage, 2008. 52–61
42 Sureka A, Jalote P. Detecting duplicate bug report using character n-gram-based features. In: Proceedings of the Asia Pacific Software Engineering Conference, Sydney, 2010. 366–374
43 Sun C, Lo D, Wang X, et al. A discriminative model approach for accurate duplicate bug report retrieval. In: Proceedings of the ACM/IEEE International Conference on Software Engineering, Cape Town, 2010. 45–54
44 Sun C, Lo D, Khoo S C, et al. Towards more accurate retrieval of duplicate bug reports. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, Lawrence, 2011. 253–262
45 Robertson S, Zaragoza H, Taylor M. Simple BM25 extension to multiple weighted fields. In: Proceedings of the ACM CIKM International Conference on Information and Knowledge Management, Washington, 2004. 42–49
46 Tian Y, Sun C, Lo D. Improved duplicate bug report identification. In: Proceedings of the European Conference on Software Maintenance and Reengineering, Szeged, 2012. 385–390
47 Banerjee S, Cukic B, Adjeroh D. Automated duplicate bug report classification using subsequence matching. In: Proceedings of the International IEEE Symposium on High-Assurance Systems Engineering, Omaha, 2012. 74–81
48 Falessi D, Cantone G, Canfora G. Empirical principles and an industrial case study in retrieving equivalent requirements via natural language processing techniques. IEEE Trans Softw Eng, 2013, 39: 18–44
49 Zhou J, Zhang H. Learning to rank duplicate bug reports. In: Proceedings of the ACM International Conference on Information and Knowledge Management, Maui, 2012. 852–861
50 Feng L, Song L, Sha C, et al. Practical duplicate bug reports detection in a large web-based development community. In: Proceedings of Asia-Pacific Web Conference on the Web Technologies and Applications, Sydney, 2013. 709–720
51 Alipour A, Hindle A, Stroulia E. A contextual approach towards more accurate duplicate bug report detection. In: Proceedings of the Working Conference on Mining Software Repositories, San Francisco, 2013. 183–192
52 Wang X, Zhang L, Xie T, et al. An approach to detecting duplicate bug reports using natural language and execution information. In: Proceedings of the International Conference on Software Engineering, Leipzig, 2008. 461–470
53 Song Y, Wang X, Xie T, et al. JDF: detecting duplicate bug reports in jazz. In: Proceedings of the ACM/IEEE International Conference on Software Engineering, Cape Town, 2010. 315–316
54 Lerch J, Mezini M. Finding duplicates of your yet unwritten bug report. In: Proceedings of the European Conference on Software Maintenance and Reengineering, Genova, 2013. 69–78
55 Kim S, Zimmermann T, Nagappan N. Crash graphs: an aggregated view of multiple crashes to improve crash triage. In: Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks, Hong Kong, 2011. 486–493
56 Dang Y, Wu R, Zhang H, et al. Rebucket: a method for clustering duplicate crash reports based on call stack similarity. In: Proceedings of the ACM/IEEE International Conference on Software Engineering, Zurich, 2012. 1084–1093
57 Bettenburg N, Premraj R, Zimmermann T, et al. Duplicate bug reports considered harmful...really? In: Proceedings of the IEEE International Conference on Software Maintenance, Beijing, 2008. 337–345
58 Wang X, Lo D, Jiang J, et al. Extracting paraphrases of technical terms from noisy parallel software corpora. In: Proceedings of the Annual Meeting of the Association for Computational Linguistics and the International Joint Conference on Natural Language Processing of the AFNLP, Singapore, 2009. 197–200
59 Cavalcanti Y, Almeida E, Cunha C, et al. An initial study on the bug report duplication problem. In: Proceedings of the European Conference on Software Maintenance and Reengineering, Madrid, 2010. 264–267
60 Cavalcanti Y, Mota Silveira Neto P, Lucrdio D, et al. The bug report duplication problem: an exploratory study. Softw Qual J, 2013, 21: 39–66
61 Davidson J, Mohan N, Jensen C. Coping with duplicate bug reports in free/open source software projects. In: Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing, Pittsburgh, 2011. 101–

108
62 Bhattacharya P, Neamtiu I. Fine-grained incremental learning and multi-feature tossing graphs to improve bug triaging. In: Proceedings of the IEEE International Conference on Software Maintenance, Timisoara, 2010. 1–10
63 Hu H, Zhang H, Xuan J, et al. Effective bug triage based on historical bug-fix information. In: Proceedings of the IEEE International Symposium on Software Reliability Engineering, Naples, 2014. 122–132
64 Lin Z, Shu F, Yang Y, et al. An empirical study on bug assignment automation using chinese bug data. In: Proceedings of the International Symposium on Empirical Software Engineering and Measurement, Lake Buena Vista, 2009. 451–455
65 Xuan J, Jiang H, Ren Z, et al. Automatic bug triage using semi-supervised text classification. In: Proceedings of International Conference on Software Engineering & Knowledge Engineering, Redwood City, 2010. 209–214
66 Alenezi M, Magel K, Banitaan S. Efficient bug triaging using text mining. J Softw, 2013, 8: 2185–2190
67 Zou W, Hu Y, Xuan J, et al. Towards training set reduction for bug triage. In: Proceedings of the Annual IEEE International Computer Software and Applications Conference, Munich, 2011. 576–581
68 Canfora G, Cerulo L. Supporting change request assignment in open-source development. In: Proceedings of the ACM Symposium on Applied Computing, Dijon, 2006. 1767–1772
69 Matter D, Kuhn A, Nierstrasz O. Assigning bug reports using a vocabulary-based expertise model of developers. In: Proceedings of the International Working Conference on Mining Software Repositories, Vancouver, 2009. 131–140
70 Xia X, Lo D, Wang X, et al. Accurate developer recommendation for bug resolution. In: Proceedings of the Working Conference on Reverse Engineering, Koblenz, 2013. 72–81
71 Baysal O, Godfrey MW, Cohen R. A bug you like: a framework for automated assignment of bugs. In: Proceedings of the IEEE International Conference on Program Comprehension, Vancouver, 2009. 297–298
72 Aljarah I, Banitaan S, Abufardeh S, et al. Selecting discriminating terms for bug assignment: a formal analysis. In: Proceedings of the International Conference on Predictive Models in Software Engineering, Banff, 2011. 12
73 Servant F, Jones J A. Whosefault: automatic developer-to-fault assignment through fault localization. In: Proceedings of the International Conference on Software Engineering, Zurich, 2012. 36–46
74 Shokripour R, Anvik J, Kasirun Z M, et al. Why so complicated? simple term filtering and weighting for location-based bug report assignment recommendation. In: Proceedings of the Working Conference on Mining Software Repositories, San Francisco, 2013. 2–11
75 Kevic K, Muller S C, Fritz T, et al. Collaborative bug triaging using textual similarities and change set analysis. In: Proceedings of the International Workshop on Cooperative and Human Aspects of Software Engineering, San Francisco, 2013. 17–24
76 Guo P J, Zimmermann T, Nagappan N, et al. Not my bug! and other reasons for software bug report reassignments. In: Proceedings of the ACM Conference on Computer Supported Cooperative Work, Hangzhou, 2011. 395–404
77 Xie J, Zheng Q, Zhou M, et al. Product assignment recommender. In: Proceedings of the International Conference on Software Engineering, Hyderabad, 2014. 556–559
78 Li W, Li N. A formal semantics for program debugging. Sci China Inf Sci, 2012, 55: 133–148
79 Gay G, Haiduc S, Marcus A, et al. On the use of relevance feedback in IR-based concept location. In: Proceedings of the IEEE International Conference on Software Maintenance, Edmonton, 2009. 351–360
80 Zhou J, Zhang H, Lo D. Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. In: Proceedings of the ACM/IEEE International Conference on Software Engineering, Zurich, 2012. 14–24
81 Deerwester S C, Dumais S T, Landauer T K, et al. Indexing by latent semantic analysis. J Amer Soc Inform Sci, 1990, 41: 391–407
82 Marcus A, Sergeyev A, Rajlich V, et al. An information retrieval approach to concept location in source code. In: Proceedings of the Working Conference on Reverse Engineering, Delft, 2004. 214–223
83 Marcus A, Rajlich V, Buchta J, et al. Static techniques for concept location in object-oriented code. In: Proceedings of the International Workshop on Program Comprehension, Louis, 2005. 33–42
84 Poshyvanyk D, Marcus A. Combining formal concept analysis with information retrieval for concept location in source code. In: Proceedings of the International Conference on Program Comprehension, Banff, 2007. 37–48
85 Poshyvanyk D, Guéhéneuc Y G, Marcus A, et al. Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. IEEE Trans Softw Eng, 2007, 33: 420–432
86 Liu D, Marcus A, Poshyvanyk D, et al. Feature location via information retrieval based filtering of a single scenario execution trace. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, Atlanta, 2007. 234–243
87 Lukins S K, Kraft N A, Etzkorn L H. Source code retrieval for bug localization using latent dirichlet allocation. In: Proceedings of the Working Conference on Reverse Engineering, Antwerp, 2008. 155–164
88 Lukins S K, Kraft N A, Etzkorn L H. Bug localization using latent dirichlet allocation. Inf Softw Technol, 2010, 52: 972–990
89 Salton G, Wong A, Yang CS. A vector space model for automatic indexing. Commun ACM, 1975, 18: 613–620
90 Rao S, Kak A. Retrieval from software libraries for bug localization: a comparative study of generic and composite

text models. In: Proceedings of the International Working Conference on Mining Software Repositories, Waikiki, 2011. 43–52
91 Chawla I, Singh S K. Performance evaluation of vsm and lsi models to determine bug reports similarity. In: Proceedings of the International Conference on Contemporary Computing, Noida, 2013. 375–380
92 Saha R K, Lease M, Khurshid S, et al. Improving bug localization using structured information retrieval. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, Silicon Valley, 2013. 345–355
93 Kim D, Tao Y, Kim S, et al. Where should we fix this bug? a two-phase recommendation model. IEEE Trans Softw Eng, 2013, 39: 1597–1610
94 Ye X, Bunescu R, Liu C. Learning to rank relevant files for bug reports using domain knowledge. In: Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering, Hong Kong, 2014. 66–76
95 Wong C P, Xiong Y, Zhang H, et al. Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis. In: Proceedings of the IEEE International Conference on Software Maintenance and Evolution, Victoria, 2014. 181–190
96 Ashok B, Joy J, Liang H, et al. Debugadvisor: a recommender system for debugging. In: Proceedings of the joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, Amsterdam, 2009. 373–382
97 Čubranić D, Murphy G C, Singer J, et al. Hipikat: a project memory for software development. IEEE Trans Softw Eng, 2005, 31: 446–465
98 Davies S, Roper M, Wood M. Using bug report similarity to enhance bug localisation. In: Proceedings of the Working Conference on Reverse Engineering, Kingston, 2012. 125–134
99 Fischer M, Pinzger M, Gall H. Populating a release history database from version control and bug tracking systems. In: Proceedings of the International Conference on Software Maintenance, Amsterdam, 2003. 23–32
100 Fischer M, Pinzger M, Gall H. Analyzing and relating bug report data for feature tracking. In: Proceedings of the Working Conference on Reverse Engineering, Victoria, 2003. 90
101 Śliwerski J, Zimmermann T, Zeller A. When do changes induce fixes? ACM Sigsoft Softw Eng Notes, 2005, 30: 1–5
102 Schröter A, Zimmermann T, Premraj R, et al. If your bug database could talk. In: Proceedings of the ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, Rio de Janeiro, 2006. 18–20
103 Zimmermann T, Premraj R, Zeller A. Predicting defects for Eclipse. In: Proceedings of the International Workshop on Predictor Models in Software Engineering, Minneapolis, 2007. 9
104 Wu R, Zhang H, Kim S, et al. Relink: recovering links between bugs and changes. In: Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering , Szeged, 2011. 15–25
105 Bissyandé T F, Thung F, Wang S, et al. Empirical evaluation of bug linking. In: Proceedings of the European Conference on Software Maintenance and Reengineering, Genova, 2013. 89–98
106 Nguyen A T, Nguyen T T, Nguyen H A, et al. Multi-layered approach for recovering links between bug reports and fixes. In: Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Cary, 2012. 63
107 Bird C, Bachmann A, Rahman F, et al. Linkster: enabling efficient manual inspection and annotation of mined data. In: Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Santa Fe, 2010. 369–370
108 Bird C, Bachmann A, Aune E, et al. Fair and balanced? Bias in bug-fix datasets. In: Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, Amsterdam, 2009. 121–130
109 Bachmann A, Bird C, Rahman F, et al. The missing links: bugs and bug-fix commits. In: Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering, Santa Fe, 2010. 97–106
110 Nguyen T H, Adams B, Hassan A E. A case study of bias in bug-fix datasets. In: Proceedings of the Working Conference on Reverse Engineering, Beverly, 2010. 259–268
111 Panjer L D. Predicting Eclipse bug lifetimes. In: Proceedings of the International Workshop on Mining Software Repositories, Minneapolis, 2007. 29
112 Giger E, Pinzger M, Gall H. Predicting the fix time of bugs. In: Proceedings of the International Workshop on Recommendation Systems for Software Engineering, Cape Town, 2010, 52–56
113 Zhang H, Gong L, Versteeg S. Predicting bug-fixing time: an empirical study of commercial software projects. In: Proceedings of the International Conference on Software Engineering, San Francisco, 2013. 1042–1051
114 Weiss C, Premraj R, Zimmermann T, et al. How long will it take to fix this bug? In: Proceedings of the International Workshop on Mining Software Repositories, Minneapolis, 2007. 1
115 Anbalagan P, Vouk M. On predicting the time taken to correct bug reports in open source projects. In: Proceedings of the IEEE International Conference on Software Maintenance, Edmonton, 2009. 523–526
116 Guo PJ, Zimmermann T, Nagappan N, et al. Characterizing and predicting which bugs get fixed: an empirical study of microsoft windows. In: Proceedings of the ACM/IEEE International Conference on Software Engineering, Cape Town, 2010. 495–504

117 Bhattacharya P, Neamtiu I. Bug-fix time prediction models: can we do better? In: Proceedings of the International Working Conference on Mining Software Repositories, Waikiki, 2011. 207–210
118 Saha R K, Khurshid S, Perry D E. An empirical study of long lived bugs. In: Proceedings of Software Evolution Week—IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering, Antwerp, 2014. 144–153
119 Ko A J, Chilana P K. Design, discussion, and dissent in open bug reports. In: Proceedings of iConference, Berlin, 2011. 106–113
120 Sahoo S K, Criswell J, Adve V. An empirical study of reported bugs in server software with implications for automated bug diagnosis. In: Proceedings of the International Conference on Software Engineering, Cape Town, 2010. 485–494
121 Xuan J, Jiang H, Ren Z, et al. Developer prioritization in bug repositories. In: Proceedings of the International Conference on Software Engineering, Zurich, 2012. 25–35
122 Bhattacharya P, Ulanova L, Neamtiu I, et al. An empirical analysis of bug reports and bug fixing in open-source Android apps. In: Proceedings of the European Conference on Software Maintenance and Reengineering, Genova, 2013. 133–143
123 Wang J, Zhang H. Predicting defect numbers based on defect state transition models. In: Proceedings of the ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, Lund, 2012. 191–200